Discrete Sequence Prediction and its Applications

Philip Laird
NASA
Ronald Saul
RECOM Technologies, Inc.

Artificial Intelligence Research Branch
NASA Ames Research Center
Mail Stop 269-2
Moffett Field, CA 94035-1000

NASA Ames Research Center

Artificial Intelligence Research Branch

Technical Report FIA-93-27
September, 1993

"To appear in Machine Learning", Vol. 13.

Discrete Sequence Prediction and its Applications

Philip Laird*
AI Research Branch

Ronald Saul†
Recom Technologies, Inc.

NASA Ames Research Center
Moffett Field, California 94035-1000 (U.S.A.)

Abstract

Learning from experience to predict sequences of discrete symbols is a fundamental problem in machine learning with many applications. We present a simple and practical algorithm (TDAG) for discrete sequence prediction. Based on a text-compression method, the TDAG algorithm limits the growth of storage by retaining the most likely prediction contexts and discarding (forgetting) less likely ones. The storage/speed tradeoffs are parameterized so that the algorithm can be used in a variety of applications. Our experiments verify its performance on data compression tasks and show how it applies to two problems: dynamically optimizing Prolog programs for good average-case behavior and maintaining a cache for a database on mass storage.

Keywords: sequence extrapolation, statistical learning, text compression, speedup learning, memory management.

Note: a preliminary version of this work appeared in (Laird, 1992a).

*Email: LAIRD@PLUTO.ARC.NASA.GOV
†Email: SAUL@KRONOS.ARC.NASA.GOV

Discrete Sequence Prediction: A Fundamental Problem

Machine learning researchers have defined a number of fundamental learning problems that occur frequently in applications. Among these are "concept" classification (learning by example to partition input vectors into two or more classes), clustering (grouping a set of items into a hierarchy of subsets whose items are related in some meaningful way), and delayed-reward associative learning (learning to associate actions with states so as to maximize the expected long-term reward). In each case a number of algorithms and architectures have been devised for solving the problem, and packaged programs implementing many of these algorithms are important elements of a growing machine-learning toolbox.

This paper adds to this list of basic learning problems that of *sequence prediction* and offers a multi-purpose toolbox algorithm for solving *discrete* sequence prediction problems (defined below).

Several different flavors of sequence prediction problems exist:

- In *discrete sequence prediction*, the input is an infinite stream of atomic symbols. The task is to find statistical regularities in the input so that the ability to predict the next symbol progresses beyond random guessing.

- In *sequence extrapolation*, the input objects may be a stream of structured objects, such as strings, integers, or trees. The task is, again, to predict the next object, but here the prediction takes the form of a rule or rules for computing the next symbol from its predecessors.

- *Time-series prediction* is distinguished from the two problems above in that time is an important element of the prediction. The task is to learn to predict the value of a function $f(t)$ for future times t based on a range of past observations. The values of the function f may be discrete or continuous.

This paper deals only with discrete sequence prediction (DSP). Humans exhibit remarkable skills in such problems, subconsciously learning, for example, that after Mary's telephone has rung three times, her machine will probably answer it on the fourth ring, or that in English the words "over and..." will probably be followed by one of the words "over", "out", or "under." Show a group of subjects the sequence

T + + + $ % T + $ % T + + + + $ %

and ask them to bet on the next symbol: there will be nearly unanimous agreement that T will follow—a remarkable concurrence, given that the symbols carry no meaning and that none of the participants is likely to have seen this particular sequence before. DSP seems to be a fundamental skill in the human learning repertory. Moreover the probabilistic nature of the predictions becomes evident if you tell the same group of subjects that the next two symbols in this sequence are T and + and ask them to predict the one after that. Now the predictions will be less confident, and most people tend to hedge by giving a best guess, say

+, along with one or two less confident alternatives. Moreover, most people can quantify approximately the amount they would be willing to bet on each alternative.

DSP plays a role in a number of applications:

- *Information-theoretic applications* rely on a probability distribution to quantify the amount of "surprise" (information) in sequential processes. For example, an adaptive file compression procedure reads through a file, treating the next character as a prediction problem and modifying its encoding for each character so that the most confident symbols are encoded with the fewest bits. Other examples are online game-playing situations where the ability to anticipate the next few moves of an opponent can increase a player's own expected score.

- *Dynamic optimization* is the task of restructuring a program, plan, database query, etc., into an equivalent one whose expected cost will be minimized on the distribution of problems that it has encountered. The learning is used, in effect, to redesign the program to be optimal on the population of problems that experience has shown the program will be asked to solve.

- *Dynamic buffering algorithms* go beyond simple heuristics like least-recently-used for swapping items between a small cache and a mass-storage device. By learning patterns in the way items are requested, the algorithm can maintain the contents of cache memory so as to minimize the cost of future data requests.

- *Adaptive human-machine interfaces* reverse the common experience whereby a human quickly learns to predict how a program (or ATM, automobile, etc.) will respond. For many years operating systems have permitted type-ahead buffering as an efficiency mechanism for humans; if the system can likewise learn to anticipate the human's responses, it, too, can work-ahead, or it can present to the user new options combining several steps, or trade speed for confidence, and the like.

- *Anomaly detection systems* identify illicit or unanticipated uses of a system. Such tasks are difficult because what is most interesting is precisely what is hardest to recognize and predict.

A number of methods are available for learning discrete sequences. Some AI researchers have approached DSP as a knowledge-based task, taking advantage of available knowledge to predict future outcomes. While a few studies have attacked sequence extrapolation/prediction directly, e.g., (Dietterich and Michalski, 1986), more often the problem has been embedded within a larger research task, as it was in (Lindsay *et al.*, 1980). Fixed-context methods predict the next symbol on the basis of some fixed number k of preceding symbols. Decision trees and neural networks can be used to make the prediction; feedforward networks, for example, have been trained to predict the pronunciation of letters from those preceding it (Sejnowski and Rosenberg, 1987).

Markov models are more sophisticated models of the input source since they capture finite-order statistical dependency. Learning an arbitrary Markov process from examples,

however, is known to be a computationally intractable problem (Abe and Warmuth, 1990; Laird, 1988), and even approximate methods based on batch algorithms may be too slow, especially for online applications, e.g., (Levinson *et al.*, 1983). *Markov trees*, however, are a related method that has been used successfully in the domain of data compression (Bell *et al.*, 1990; Blumer, 1990; Williams, 1988). These structures allow the context model for the next symbol to vary, and perhaps in part for this reason they have been found empirically to be more effective for text compression than Markov process and dictionary models. Their most significant drawback is that they tend to command large amounts of storage, to the extent that good ways have been sought to trade storage for modeling power (Lelewer and Hirschberg, 1991).

The TDAG algorithm, presented below, is based on the Markov-tree approach. By viewing it as a learning algorithm, and taking advantage of current techniques for analyzing learning algorithms, we are in a position to customize it to take on a rather broad range of applications. For example the algorithm is parameterized so as to control the most important performance parameters:

- the turnaround time (time between arrival of an input symbol and the return of the next prediction);
- space utilization; and
- the rate of adaptation to changes in the input process.

This paper is organized as follows. First we give the TDAG algorithm—more accurately, several flavors of the algorithm. Then we analyze its time and space requirements and prove a correctness result. Finally we describe three experiments where the TDAG algorithm served well as the adaptive learning element.

The TDAG Algorithm

TDAG (Transition Directed Acyclic Graph) is a sequence-learning tool. It assumes that the input consists of a sequence of discrete, uninterpreted symbols and that the input process can be adequately approximated in a reasonably short time using a small amount of storage. That neither the set of input symbols nor its cardinality need be known in advance is a feature of the design. Another is that the time required to receive the next input symbol, learn from it, and return a prediction for the next symbol can be tightly controlled and in most applications, bounded.

For perspicuity we present the TDAG algorithm by successive refinement, beginning with a very simple but impractical learning/prediction algorithm and subsequently repairing its faults. First, however, let us provide some intuition for the basic Markov-tree algorithm.

Assume that we have been observing input symbols for some time and that we want to

input(x): `/* x = the next input symbol */`

- Initialize `new-state` := $\{\Lambda\}$.
- For each node ν in `state`,
  - Let μ := *make-child*(ν, x). `/*` (See below) `*/`
  - Enqueue μ onto `new-state`.
- `state` := `new-state`.

make-child(ν, x): `/* create or update the child of` ν `labeled` x `*/`

- In the list `children`(ν), find or create the node μ with a `symbol` of x. If creating it, initialize both its count fields to zero.
- Increment `in-count`(μ) and `out-count`(ν) each by one.

project-from(ν): `/* Return a probability distrib.*/`

- Initialize `projection` := $\{\}$.
- For each child μ in `children`(ν), add to `projection` the pair [`symbol`(μ), (`in-count`(μ)/`out-count`(ν))].
- Return `projection`.

Figure 1: Basic algorithms for new-symbol input and for projection.

time, "`i`" 15% of the time, "`r`" and "`a`" each 10%, and a few others with smaller likelihoods. This can form the basis for a probabilistic prediction of the next symbol and its likelihood. Alternately we could base such a prediction on just the previous *two* characters "`th`", on the preceding character "h", or on none of the preceding characters by just counting symbol frequencies. Or we could form a weighted combination of all these predictions. If both speed and accuracy matter, however, we will probably do best to base it on the strongest conditioning event "` th`", since by assumption it has occurred enough times for the prediction to be confident.

The Basic Algorithm. TDAG learns by constructing a tree. Initially the tree consists of only the root node, Λ; with the arrival of each symbol one or more new nodes are added. Stored with each node is the following information:

- `symbol` is the input symbol associated with the node. For the root node, this symbol is undefined.
- `children` is a list of the nodes that are successors (children) of this node.

- `in-count` and `out-count` are counters, explained below.

If ν is a node, the notation **symbol**(ν) means the value of the **symbol** field stored in the node ν, and similarly for the other fields. There is one global variable, **state**, which is a FIFO queue of nodes; initially **state** contains only the node Λ. For each input symbol x the learning algorithm, *input* (Figure 1), is called. We obtain a prediction by calling *project-from* and passing as an argument the last node ν on the **state** queue for which out-count(ν) is "sufficiently" high, in the following sense.

The `in-count` field of a node ν counts the number of times that ν has been placed on the **state** queue. This occurs if **symbol**(ν) is input while its parent node is on the **state** queue, and we say that ν "has been visited" from its parent. The `out-count` field of a node ν counts the number of times that ν has been replaced by one of its children on the **state** queue. If μ is a child of ν, the ratio `in-count`(μ)/`out-count`(ν) is the proportion of μ visits among all visits from ν to its children. It is equally an empirical estimate of the probability of a transition from visiting the node ν to visiting its child μ. If we treat transitions from ν as occurring independently, the statistical confidence in this probability increases exponentially as `out-count`(ν) increases, so we can use a minimum value for the `out-count` value as a basis for deciding on which node to base our prediction.

There are two ways to interpret the meaning of a "node": as a string of symbols (specifically, the symbols along the path from the root to the node), and more generally as a context on which to make a prediction. The latter view is especially useful in applications, where more information is often stored at a node than just the input symbol.

As a simple example (Figure 2), suppose the string "**ab**" is input to an empty TDAG. The result is a TDAG with four nodes:

- Λ, the root, with two children. The `out-count` is two.
- **a**, the child of Λ labeled `a`, created upon arrival of the symbol `a`. This node has been visited only once, so its `in-count` is 1. Its only child has been visited once (with the arrival of the `b`), so its `out-count` is also 1.
- **b**, the child of Λ labeled `b`, created upon arrival of the symbol `b`. Its `in-count` is 1, but since no character has followed `b`, it has no children and its `out-count` is 0.
- **ab**, the child of the node **a**. It was created upon arrival of the symbol `b`, so its **symbol** is `b`. The `in-count` is 1, and the `out-count` is 0.

The **state** queue now contains three nodes: Λ, **b**, and **ab** (in that order). These nodes represent the three possible conditional events upon which we can base a prediction for the next symbol: Λ, the null conditional event; **b**; the previous symbol `b`; and **ab**, the two previous symbols. If we project from Λ, the resulting distribution is [(`a`, 1/2), (`b`, 1/2)]. We cannot yet project from either of the other two nodes because both nodes are still leaves. Our confidence in the projection from Λ is low because it is based on only `out-count`(Λ) = 2 events. However, `out-count`(Λ) increases linearly with the arrival of input symbols, so that we shall not have to wait long to be able to make a quite reasonable prediction.

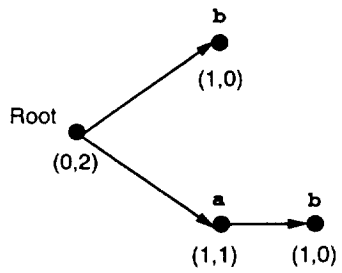

Figure 2: TDAG tree after "ab" input. The numbers in parentheses are, respectively, the `in-count` and `out-count` for the nodes.

The basic algorithm is impractical: the number of nodes on the **state** queue grows linearly as the input algorithm continues to process symbols, and as a simple exercise one can show that the rate of growth of the total number of nodes in the TDAG graph may be quadratically related to the number of input symbols. The trick, then, will be to restrict the use of storage without corrupting or discarding the useful information.

The *turnaround time* for the procedure *input* to process each new symbol x is proportional to both the size of **state** and the time to find the appropriate child of each **state** node (*make-child*, step 1). The **state** size is constantly growing, so as written, the turnaround will take longer after each symbol. The improvements below will rectify this problem. The time to find the child node is also unbounded because there is no limit on the number of distinct input symbols and hence on the length of the `children` list. We can reduce the search for the appropriate child to virtually constant time by using a hash table indexed by the node address ν and the symbol x to obtain the address μ of the successor of ν labeled by x.

The Improved Algorithm. To render the basic algorithm practical, we shall make three important modifications. Each change is governed by a parameter and requires that some additional information be stored at each node. It is also convenient to maintain a new global value, `sym-count` that increases by one for every input symbol. The changes are:

- *Bound the node probability.* We eliminate nodes in the graph that are rarely visited, since such nodes represent symbol strings that occur infrequently. More accurately, we establish a minimum threshold probability Θ and *refuse to extend (adjoin children to) any node whose probability of being visited on any round is below this threshold.*

- *Bound the height of the TDAG graph.* The previous change in itself tends to limit the height of the TDAG graph, since nodes farther from the root occur less often on average. But there remain input streams that can cause unbounded growth of the graph (one example is "`aaa ...`"). As a precaution, therefore, we introduce the parameter **H** and refuse to extend any node ν such that `height`(ν) equals this threshold.

- *Bound the prediction size.* In the worst case the time for *project-from* to compute a projection is proportional to K, the number of distinct symbols in the input. This is unacceptable for some real-time applications since K is unknown and in general may be quite large. Thus we limit the size of the projection to at most `max-projection` symbols, `max-projection` ≥ 1. Doing so means that any symbol whose empirical likelihood is at least $1/$`max-projection` will be listed with its likelihood as part of the projection.

The first change above is the most difficult to implement since it requires an estimate of Prob(ν), the probability that ν will be visited on a randomly chosen round. We shall assume that, for any node ν, the probability Prob(ν) exists. Actually Prob(ν) may not exist unless the input is generated by a stationary process, and the event that ν is visited on a given round is not an independent sequence of random variables. Still, the simplifying assumption that Prob(ν) exists and can be estimated by sampling is useful for the insight and for the numerical design guidelines we obtain.

Pending this estimate, we can adopt an *eager strategy* by extending a node until statistics indicate that Prob(ν) $< \Theta$ and then deleting its descendents, or a *lazy strategy* by refusing to extend a node until sufficient evidence exists that Prob(ν) $\geq \Theta$, or some intermediate strategy. After sufficiently many input symbols have been processed, the eager and the lazy strategies result ultimately in the same model. The eager strategy temporarily requires more storage but produces better predictions during the early stages of learning.

Note that in the basic algorithm the `state` always contains exactly one node of each height $h \leq$ `sym-count`, where `sym-count` is the number of input symbols so far. Let ν be a node of height h; with some reflection it is apparent that, if `sym-count` is $\geq h$, then the fraction of times that ν has been the node of height h on `state` is `in-count`(ν)/(`sym-count`$-h+1$). We denote this by Prob(ν | `sym-count`). Moreover, as `sym-count` $\rightarrow \infty$, Prob(ν | `sym-count`) approaches Prob(ν) if this limit exists. Since the decision about ν's extendibility must be made in finite time (indeed, as soon as justifiably possible), we establish a parameter **N** and make the algorithm wait for **N** input symbols (transitions from the root) before deciding the extendibility of nodes of height 1 (those immediately following the root node).

Thereafter, another 2**N** input symbols are required before nodes of height 2 are decided, 3**N** for nodes of height 3, and so forth: h**N** symbols are needed to decide nodes of height h after having decided the extendibility of those of height $h-1$. The reason we wait for more transitions before estimating Prob(ν) for nodes of greater height is that the number of TDAG nodes with height h may be an exponential function of h, and a sample size linear in h helps maintain a minimum confidence in our decision about all nodes in the TDAG, regardless of their height.

We can justify this sample size a bit more formally. Statistical confidence is the probability that an estimator for a statistic will be within some range $\pm\epsilon$ of its true value. A statistical fact is that when we flip a b-sided coin N times and estimate the probability of each side by the relative frequency of its occurrence, our confidence in the accuracy of the estimates increases toward 1 exponentially with N, independent of b and ϵ. Let us choose a value $0 < \delta < 1/2$ and let N be chosen so that, after observing N transitions to its

input(x): `/* process one input symbol */`

- `sym-count` := `sym-count` + 1. Initialize `new-state` := $\{\Lambda\}$.
- For each node ν in `state`,
  - Let $\mu :=$ *make-child*(ν, x). [See below.]
  - If *extendible?*(μ), then enqueue μ onto `new-state`.
- `state` := `new-state`.

make-child(ν, x): `/* find or create a child node of` ν `*/`

- In the list `children`(ν) find or create the node ν with `symbol`(μ) = x. If creating it, initialize: `in-count`(μ) and `out-count`(μ) := 0, `height`(μ) := `height`(ν) + 1, `extendible-p`(ν) := *unvalued*, and `children`(μ) and `most-likely-children`(μ) := *empty*.
- Increment `in-count`(μ) and `out-count`(ν) each by one.
- Revise the (ordered) list `most-likely-children`(ν) to reflect the increased likelihood of μ.

Figure 3: Revised input algorithm.

successors, confidence in the accuracy of the transition probabilities for the root node is at least $1 - (\delta/(1+2\delta)) \equiv 1 - \delta'$. The TDAG algorithm uses $(h+1)N$ transitions to estimate transition probabilities from nodes of height h (for $h = 0, 1, \ldots$). Since there are $\mathcal{O}(2^h)$ nodes at height h, the total probability that even one of the transition probability estimates for a node at height h is unacceptably inaccurate is no more than $\mathcal{O}(2^h(\delta')^{h+1})$. Over the entire tree, therefore, this probability is at most

$$\begin{aligned} & \mathcal{O}(\delta' + 2(\delta')^2 + \ldots + 2^h(\delta')^{h+1} + \ldots) \\ = \; & \mathcal{O}(\delta'/(1-2\delta') \\ = \; & \mathcal{O}(\delta) \end{aligned}$$

In this calculation we have assumed that the transition from one node is approximately independent of transitions from its predecessors. There is no reason to believe this is the case, but again it serves to obtain useful estimates.

Note that, under this policy, all nodes of height h conveniently become decidable after the arrival of $N(1 + 2 + \ldots + h) = Nh(h+1)/2$ symbols. For the applications described in this paper a node marked extendible or unextendible remains so thereafter even if later the statistics seem to change. A switch **extendible-p** is stored with each node. It remains unvalued until a decision is reached as to whether ν is extendible, and then is set to **true**

if and only if ν is extendible. This policy of deciding once whether a node is extendible or not is a deliberate compromise for efficiency: for the applications described in this paper the statistics of the source remain stable for the duration of the problem. In applications where such is not the case, however, the TDAG structure must remain more flexible. It is straightforward to modify the algorithm to recheck the node probabilities periodically and change the extendibility property of nodes as appropriate.

The revised input algorithm is shown in Figure 3. The lazy and eager versions of the *Extendible?* routine are given in Figure 4.

In the prediction algorithm, we store in each node a list called `most-likely-children` containing the most likely children of that node, numbering up to `max-projection`. Whenever an input symbol causes a node ν to be replaced by one of its children μ in the `state`, we adjust the list of ν's most likely children to account for the higher relative likelihood of μ. This can be done in time $\mathcal{O}$(`max-projection`). The algorithm is in Figure 5.

Analysis

Time and Space Requirements. The time and space requirements of the algorithm are governed entirely by the four user parameters $\mathbf{H}$ (the maximum height), $\mathbf{\Theta}$ (the minimum node probability), $\mathbf{N}$ (the number of input values for estimating $\mathbf{\Theta}$), and `max-projection` (the maximum number of symbols in a projection). For an online algorithm, an important quantity is the *turnaround time*: the time to process each input symbol. Turnaround time for the input algorithm is governed by the number of nodes on the `state` stack (at most $\mathbf{H}$), the time to locate a child node with the current input symbol ($\mathcal{O}(1)$ with the appropriate hashing), and the time to increment counters ($\mathcal{O}(\log$ `sym-count`$)$).[1] In the worst case, the discrete complexity is $\mathcal{O}(\mathbf{H} \cdot \log$ `sym-count`$)$. In many practical cases, the input source does not suddenly and radically change its statistical characteristics, and as a result there is no need for periodic re-evaluation of the extendibility of the TDAG nodes. Consequently the graph "freezes" once the leaf nodes have all been marked unextendible. This occurs after at most $1 + (\mathbf{NH}(\mathbf{H}+1)/2)$ input symbols (but usually much sooner). In such situations, the counters can also be frozen, and the discrete complexity is only incrementing the counters, and the $\mathcal{O}(\mathbf{H})$.

Controlling the space requirements of an on-line algorithm is crucial. For the lazy extension strategy (which is the more conservative with respect to storage), total number of TDAG nodes is at most $K(1+\mathbf{H}/\mathbf{\Theta})$, where K is the size of the input alphabet. To see this, note that at each height ≥ 1 there can be at most $1/\mathbf{\Theta}$ extendible nodes. Hence, counting the root node, the number of extendible nodes is at most $(1+\mathbf{H}/\mathbf{\Theta})$. And since extendible nodes may have up to K children that are not extendible, the claim follows. Pointers from nodes to their children account for only a constant storage factor since the graph is a tree. If the count fields, however, are allowed to increase without bound, the total storage requirement increases as log `sym-count`.

[1] Over a sequence of `sym-count` input symbols, the total of all the counters in the tree is $\mathcal{O}$(`sym-count`). Hence when amortized over the entire sequence, this log `sym-count` factor becomes $\mathcal{O}(1)$. For real-time applications, however, amortized complexity is not an appropriate complexity measure.

Extendible?(μ): `/*''Lazy'' Version */`

- If `extendible-p`(μ) is valued (`True` or `False`), return its value.
- Else let h = `height`(μ); if `sym-count` $\leq \mathbf{N}h(h+1)/2$, then return `False`. (μ is still undecided).
- If `height`(μ) = $\mathbf{H}$ (i.e., μ is at the maximum allowed height) or (`in-count`(μ) $- 1$) $< \mathbf{\Theta} \cdot h\mathbf{N}$ (i.e., Prob(μ) is below threshold), then
  - `extendible-p`(μ) := `False`.
  - Return `False`.
- Else
  - `extendible-p`(μ) := `True`.
  - Return `True`.

Extendible?(μ): `/*''Eager'' Version */`

- If `extendible-p`(μ) is valued (`True` or `False`), return its value.
- Else let h = `height`(μ); if `sym-count` $\leq \mathbf{N}h(h+1)/2$, then return `True`. (μ is still undecided).
- If `height`(μ) = $\mathbf{H}$ (i.e., μ is at the maximum allowed height) or (`in-count`(μ) $- 1$) $< \mathbf{\Theta} \cdot h\mathbf{N}$ (i.e., Prob(μ) is below threshold), then
  - Remove all nodes below μ.
  - Reinitialize: `extendible-p`(μ) := `False`, `children`(μ) := *empty*, and `most-likely-children`(μ) := *empty*.
  - Return `False`.
- Else
  - `extendible-p`(μ) := `True`.
  - Return `True`.

project-from(ν):

- Initialize `projection` := {}.
- For each child μ in `most-likely-children`(ν), add to `projection` the pair [`symbol`(μ), `in-count`(μ)/`out-count`(ν)].
- Return `projection`.

Figure 5: Revised prediction algorithm

The *prediction time*—the time required to return a list of up to `max-projection` symbols and their estimated probabilities—depends upon the time to choose a node from which to project ($\mathcal{O}(\mathbf{H})$, as discussed below), and the time to compute the projection using *project-from*. With the efficiencies introduced for just this purpose, the prediction time is $\mathcal{O}(\mathbf{H} + \texttt{max-projection} \cdot \log(\texttt{sym-count}))$; and as above, the $\mathcal{O}(\log \texttt{sym-count})$ factor is often removable in practice if we freeze the TDAG.

Correctness Properties. It is easy to construct a fast sequence prediction algorithm if there are no requirements governing the quality of the predictions—any old guess will do. Evidently some notion of "correctness" is needed to ensure that the predictions have a well-defined meaning and to permit comparison with other algorithms. But unlike complexity, correctness is an extensional property that cannot be discussed solely by examining the algorithm; we must make some assumptions about the family of input sources.

TDAG makes a simple, efficient model of the source; ultimately its value as a sequence prediction algorithm will depend upon its ability to produce *good* models of actual sources. One family of good models are stochastic deterministic finite automata (SDFAs). SDFAs are stochastic processes that model Markovian dependencies between events—where the value of one random variable depends on the values of only a bounded number of other random variables. An SDFA is a deterministic finite automaton with a probability distribution assigned to the transitions from each node. In Figure 6, for example, we find a two-state SDFA over a two-symbol alphabet; the transitions from state 1 are assigned probabilities 9/10 for `a` and 1/10 for `b`.

The problem of "learning" an SDFA from examples of the sequences it generates is, in more than one sense, intractable with respect to polynomial time computation (Abe and Warmuth, 1990; Laird, 1988), and while methods exist for finding approximate SDFAs in special cases, the author is unaware of an algorithm for learning SDFA models that is practical enough for online applications.

TDAG models are closely related to SDFA models, as we shall demonstrate. They are

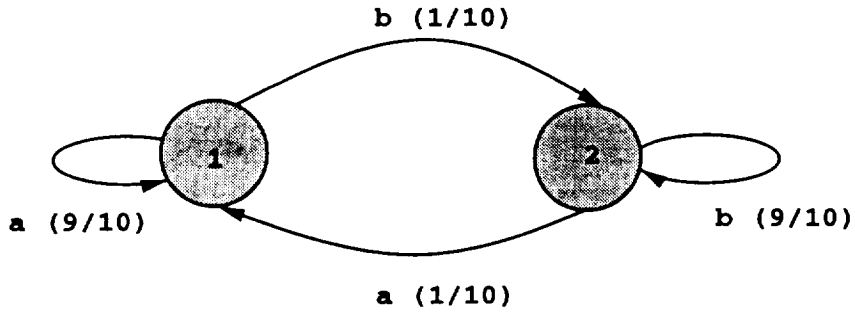

Figure 6: A simple SDFA

real sources, it stands to reason that TDAGs are also good models of many real sources.

The TDAG represents the SDFA source as a "Markov tree": the root node represents the steady state, depth-one nodes represent the possible one-step transitions from steady state, etc. This relationship is more clearly explained by example. When the SDFA in Figure 6 is in state 1, the process generates a run of zero or more a's (averaging 9 in number), until it produces b and changes to state 2. State 2 is similar to state 1, except that the roles of a and b are reversed. Suppose that this process has been running for a while without our watching it, so that we have no information about its current state. By symmetry it is clear that the probability that the automaton is in state 1 is 1/2 and 1/2 for state 2. Hence the likelihood of its next generating an a is $(1/2)(9/10) + (1/2)(1/10) = 1/2$, and the same for b.

The conditional probabilities of the subsequent characters are also easy to calculate. For example, if told that the most recent character emitted by the process was b, we infer that the process is in state 2 with probability one. Consequently the probability of next generating a is 1/10, versus 9/10 for b. More generally, information about what characters an SDFA has produced recently affects the probabilities of what state it is in and therefore what character it will emit next.

The first few levels of the probability tree for this SDFA are shown in Fig. 7. The root node corresponds to the lack of any information about the state of the SDFA, and each path through the tree represents a unique sequence of output symbols corresponding to the labels on the nodes of the trees. Edges are labeled with transition probabilities. The meaning of the encircled probability 1/2 is: *in the absence of any information about the state of the SDFA, the likelihood is* 1/2 *that the next character is* b. Similarly, the enboxed probability 1/10 means: *given only that the process has just produced a* b *(but no information about what it produced before that), the next character will be an* a *with probability* 1/10.

In this example, the probability tree is very simple in that all nodes below the root have the same probability distribution. In general, however, the probability tree for an SDFA may be different at each level, requiring an infinite tree to represent all contexts.

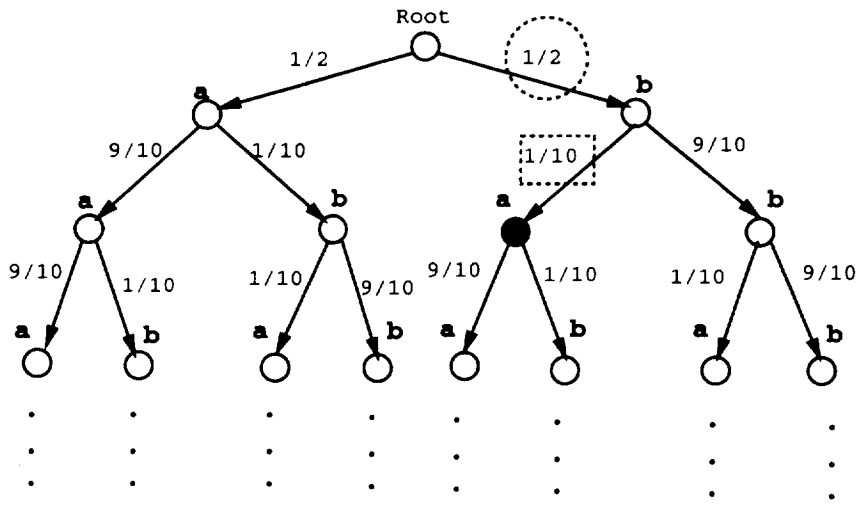

Figure 7: Part of the steady-state probability tree for the SDFA of Fig. 2

Imagine now that we feed the symbol stream from this SDFA into a TDAG, let it run for a long time, and then examine the resulting TDAG tree. For now let us use the Basic Algorithm without any of the refinements. We shall find that, if we relate the edge probabilities on the probability tree to the corresponding `out-count/in-count` estimate for the TDAG transition probabilities, the TDAG tree converges in some sense to the tree in Fig. 7. For example, whenever the SDFA emits an **a**, the TDAG *input* algorithm increases `out-count` for the root and `in-count` for its "**a**" child node ν. The ratio `in-count`(ν)/`out-count`(Λ) is the proportion of "**a**"s input to the TDAG, just as the encircled probability in Fig. 7 is the probability that the process emits an **a**. As time continues, the finite, but growing, TDAG tree becomes a better approximation to the (infinite) probability tree.

Next, consider projection. Suppose the SDFA generates the sequence "**aba**," and we want to predict the next character. If we have computed the SDFA probability tree only to a depth of 3, we cannot use the tree directly to predict the next character using the full three-character conditioning event. It is possible, however, to ignore the first **a** and use only the last two characters ("**ba**") as our conditioning event to make a prediction; this corresponds to making the prediction for the SDFA knowing only the past two symbols it has generated. In exactly the same way, if the TDAG is available only to a height of three, the next character can still be projected based upon the node that is shaded black in Fig. 7. This node would be the one most recently placed on the **state** queue, and if *project-from* were called with this node as argument, the projection would consist of **a** with probability 9/10 and **b** with probability 1/10.

This example is based on a very simple SDFA, but the idea extends formally to an arbitrary SDFA, as follows. Let S be an SDFA with n states over the alphabet Σ. For

convenience we may assume that S is closed—i.e., every state is reachable with positive probability from every other state. The (infinite) probability tree for S is computed as follows.

- Let π_i ($i \in \{1, \ldots, n\}$) be the unique steady-state probability distribution for the states of S. π_i represents the limiting value of the fraction of all transitions that enter state i.
- Label the root of the probability tree with the symbol Λ and associate with it the vector $\pi \equiv [\pi_1, \ldots, \pi_n]$.
- Recursively, let ν be a node in the tree associated with the state distribution vector π'. For each symbol $x \in \Sigma$ create a descendent node μ labeled x. Label the edge from ν to μ with the probability: $P(x \mid \pi') = \sum_{i=1}^{n} \pi'_i P(x \mid i)$, where $P(x \mid i)$ is the probability that the SDFA state i generates an x. Also, associate with the node μ the distribution

$$\pi''_j = \frac{\sum_{i=1}^{n} \pi'_i P(x, j \mid i)}{P(x \mid \pi')},$$

  where $P(x, j \mid i)$ is the probability that S in state i generates an x and moves to node j, given that the current state is i.

Suppose the stream of symbols generated by S is sent to a TDAG (basic version). Let us argue that this TDAG approaches the probability tree in the limit. For transitions from the root node Λ to its descendent ν labeled with the symbol x, the ratio $\texttt{in-count}(\nu)/\texttt{out-count}(\Lambda)$ for the TDAG is, as noted above, the fraction of x occurences among all symbols output by S; in the limit, this approaches $P(x \mid \pi)$. Hence the edges from the root of the TDAG to its immediate descendents correspond in the limit to the edges out of the root node of the probability tree. Consider next the edge from this depth-one node ν to its depth-two child μ labeled with the symbol y. The algorithm places μ on **state**

whenever the pair of symbols xy occur consecutively; hence after N symbols, the fraction f of TDAG input symbols following which μ has been placed on **state** is

$$\begin{aligned} f &= \frac{\texttt{in-count}(\mu)}{\texttt{out-count}(\Lambda)} \\ &= \frac{\texttt{out-count}(\nu)}{\texttt{out-count}(\Lambda)} \cdot \frac{\texttt{in-count}(\mu)}{\texttt{out-count}(\nu)} \\ &\rightarrow \frac{\texttt{in-count}(\nu)}{\texttt{out-count}(\Lambda)} \cdot \frac{\texttt{in-count}(\mu)}{\texttt{out-count}(\nu)} \quad \text{as} \quad N \rightarrow \infty. \end{aligned}$$

But we have seen that $\texttt{in-count}(\nu)/\texttt{out-count}(\Lambda) \rightarrow P(x \mid \pi)$, and from the SDFA side, we have that

Putting these together, we find that

$$\frac{\texttt{in-count}(\mu)}{\texttt{out-count}(\nu)} \rightarrow P(y \mid \pi')$$

in the limit.

Continuing this argument down the tree for nodes of increasing depth, we conclude:

> *For any finite, closed, discrete-time SDFA source S and for any integer $d \geq 0$, the portion of the TDAG generated by the* input *algorithm of Figure 1 consisting of all nodes of depth $\leq d$ converges with probability one to the portion of the Markov probability tree for S consisting of all nodes of depth $\leq d$.*

It would be nice to give a polynomial bound on the number of input symbols needed before the probabilities $\texttt{in-count}(\mu)/\texttt{out-count}(\nu)$ are within ϵ of their true values, for all nodes of some fixed depth. Unfortunately, however, the number of symbols required may be exponential in the number of states of the underlying SDFA and in the size of the alphabet (Laird, 1988). The best we can say is that these ratios are maximum-likelihood estimators for the true values—i.e., after any finite number of input symbols, they maximize the likelihood of the observed distribution of symbols, conditional upon the context represented by the node ν—and are in that sense optimal for the information available.

Consider now what effect the refinements to the basic TDAG algorithm have on the correspondence between the SDFA and TDAG trees. The height bound **H** on the TDAG obviously corresponds to truncating the tree by removing all nodes at a distance of more than **H** from the root. But instead of simply shearing off all branches uniformly at a fixed height, the algorithm retains more nodes along branches that are most frequently traversed while cutting back the less probable paths. As we have seen above, the parameters relate directly to the available computational resources (space and time), rather than to unobservable quantities like the hypothetical number of states in the source process. The refinement induced by the Θ parameter corresponds to removing from the SDFA tree all descendents of any node whose node probability is less than Θ. Finally, limiting the size of a projection to at most `max-projection` affects, not the tree, but the way we use it to make predictions.

Experiments

In order to test the usefulness of TDAG in actual settings, we chose three prototypical applications wherein TDAG served as the learning component. In this section we describe each

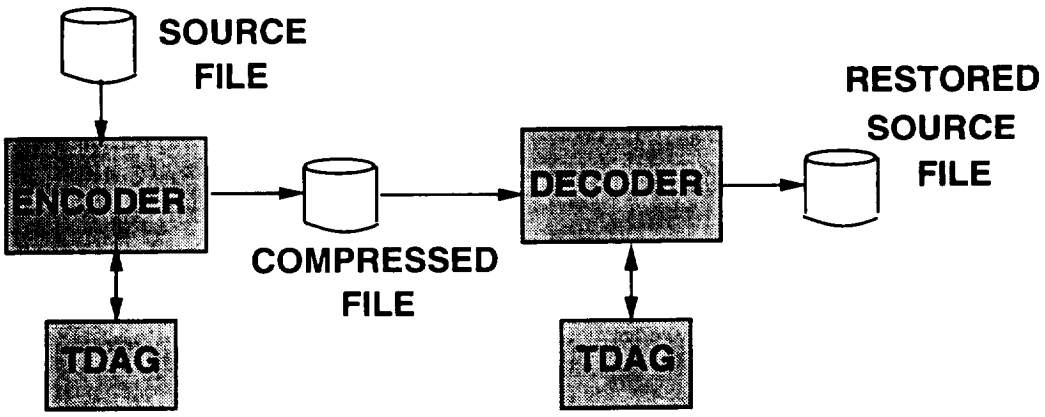

Figure 8: Data Compression using TDAG

(Lelewer and Hirschberg, 1987) maps a probability distribution for the next symbol into a code, using Huffman's algorithm to assign the fewest bits to the most probable characters. Arithmetic coding is another commonly used method to translate probabilities into a code; although it often yields shorter codes, its algorithm is more difficult to program, and so we chose the Huffman alternative.

The TDAG serves nicely as the learning element in an adaptive compression program: each character is passed to the TDAG input routine and a prediction is returned for the next character. This prediction is used to build or modify a Huffman code, which is attached to each extendible node in the TDAG.

The inverse procedure decompresses the file: a Huffman code based on the prediction for the next character is used to decode the next character; that character then goes into the TDAG, whereupon the Huffman code for the subsequent character is returned. (See Figure 8.) Note that the TDAG constructed by the encoder is *not* part of the compressed file or passed to the decoder: the decoder reconstructs the TDAG as it decodes the compressed text.

For compressing ASCII text, the TDAG parameters were set as follows: a maximum depth $\mathbf{H} = 15$ (though the actual graph never reached this height); `max-projection` $= 120$ (since no more than 120 characters actually occur in normal ASCII text files); minimum node probability $\mathbf{\Theta} = 0.002$; and extendibility sample size $\mathbf{N} = 10$. An eager version of the extendibility predicate worked best.

Predictions from a node will not always include a positive probability for every character. For this reason the TDAG was initialized with a default distribution in the root node by creating a child node with a `in-count` value of 1 for every valid character. If during compression a character occured that was not present in the prediction, an "escape" code was generated, and the root node distribution was used to encode the character. ((Bell *et*

The utility of a text compression utility depends on both its compression and its efficiency. In this case there was no need for an efficient implementation since the purpose was merely to check the quality of the TDAG predictions. We measured the compression by the *compression ratio*, defined as the ratio of the file size after compression to that before compression (so that smaller values are better). The simple TDAG text compression program, implemented in about an hour in LISP, yielded compression ratios considerably better than those for the `compact` program (FGK algorithm) and, except for small files, better than those of the benchmark UNIX[2] `compress` utility (LZW algorithm). Results for files in three languages are shown in Figure 9.

File Size (bytes)	Language	Compression (%)		
		TDAG	`compress`	`compact`
4301	C	46.1	45.9	60.6
13334	English	51.7	53.2	60.3
70101	Lisp	33.4	38.0	41.5

Figure 9: Sample File Compression Results. "Compression" is the compressed length divided by the original length.

Dynamic Optimization *Dynamic optimization* is the task of tuning a program for average-case efficiency by studying its behavior on a distribution of typical problems. Sequences of predictable computational steps can be partially evaluated and unfolded into the program. Also, choices that entail search can be ordered so as to minimize the search time. Any program transformations, however, must result in a program that is semantically equivalent to the original.

This work is based on the work of many researchers, especially that of Prieditis and Mostow (Prieditis and Mostow, 1987) on PROLEARN and of Gooley and Wah (Gooley and Wah, 1989) on Markov models of Prolog execution. Recently a number of other studies have sought ways to improve program performance by uncovering statistical regularities, e.g., (Gratch and DeJong, 1992; Greiner and Orponen, 1991; Subramanian and Feldman, 1990).

As a TDAG application we implemented a new kind of dynamic optimizer for Prolog programs using a TDAG as the learning element. Details of the implementation are given elsewhere (Laird, 1992b), along with a comparison to other methods. Here we summarize the essential ideas.

Adapting a DSP algorithm to perform dynamic optimization is non-trivial because prediction is only part of the problem. If several choices are possible, we must balance the likelihood of success against its cost. In repairing a car, for example, replacing a cheap, easily-installed part may be less likely to fix the problem than replacing a more expensive

[2]UNIX is a trademark of AT&T Bell Laboratories.

one, but may still be the repair to try first if the ratio of cost to probability of success is smaller. So in addition to tracking which program events follow which, we need to measure how much each choice costs.

Logic programs represent search problems in which the task is to find a clause $[C] : H \leftarrow T_1, T_2, \ldots$ whose head H unifies with the input goal and whose subgoals T_i (after applying a unifying substitution) are all refutable. If we can predict with certainty which clause should be chosen for any given goal, then the cost of running the program is linear in the size of the solution. Lacking a way to make such predictions perfectly, we instead use a TDAG to guide us to the right clauses during the proof. If there are n clauses available for attempting to solve a goal, the "right" clause is the one which maximizes the ratio of the likelihood of success to the expected cost of using the clause. Naturally, which clause that is depends on the context within the proof (computation).

If we can predict the optimal clause order, we can then transform the program so that optimal choices are made automatically. In the Prolog language[3] we can apply the following three program transformations without changing the semantics of the program: (see Fig. 10)

- (Unrolling) Copy all clauses of a predicate `p` and assign the predicate a new name, e.g., `pcopy`. Some references to the old predicate `p` in the tail literals of `p` clauses can be changed to call to `pcopy` instead.
- (Reordering) Reorder the clauses of a particular predicate.
- (Unfolding) Unfold two clauses C_1 and C_2 by resolving a tail literal from C_2 with the head of C_1. The result is a new clause to be added to the program.

These are not the only possible transformations for logic programs, but they are sufficient to obtain good results in practice and simple enough to understand mathematically. Mathematical prediction of the expected effect of each transformation on the performance of the program is essential in this approach.

The system we devised has two phases: a learning phase and an optimization (transformation) phase. (See Figure 11.) During the learning phase a TDAG gathers data about the probabilities and costs of success and failure of specific clauses at specific points (contexts) in the proof. Training examples are drawn from a representative source of problems that we assume is statistically similar to the actual production problems for which the optimized program will be used.

TDAG nodes are labeled with program clauses. When a clause $[C] : H \leftarrow T_1, \ldots$ is invoked, a transition in the TDAG structure is recorded from C to each clause for the predicate T_1 that is invoked during the computation. The `in-count` count for the clause is separated into two parts: invocations that eventually succeed and invocations that eventually

Note: In the programs below only the predicates are shown, without their arguments—e.g., p instead of p(...).

(a) Initial program:

```
[C1]:  p <- p.
[C2]:  p.
```

(b) After an unrolling: (Clauses C_3 and C_4 are copies of C_1 and C_2, resp., but with the predicate renamed.

```
[C'1]:  p <- pcopy.
[C2]:   p.
[C3]:   pcopy <- pcopy.
[C4]:   pcopy.
```

(c) After reordering the pcopy predicate:

```
[C'1]:  p <- pcopy.
[C2]:   p.
[C4]:   pcopy.
[C3]:   pcopy <- pcopy.
```

(d) Let θ be a unifier for the underscored literals above; after unfolding C'_1 through C_4, we obtain:

```
[C'1.1]:  pθ.
[C2]:     p.
[C'1.2]   p <- pcopyrest.
[C4]:     pcopy.
[C3]:     pcopy <- pcopy.
[C3.1]:   pcopyrest <- pcopy.
```

Here, $C_{3.1}$ is a copy of C_3; note that the case C_4 is covered by the unfolded clause $C'_{1.1}$.

Figure 10: Basic Prolog transformations

clause, regardless of success or failure. With a reasonable training sample, therefore, the TDAG returns good statistics on the probability that the clause will succeed if invoked in various contexts, and the expected cost of invoking the clause. These statistics are sufficient for predicting the effectiveness of unfolding and reordering transformations.

The optimization phase uses the TDAG statistics to select transformations that are

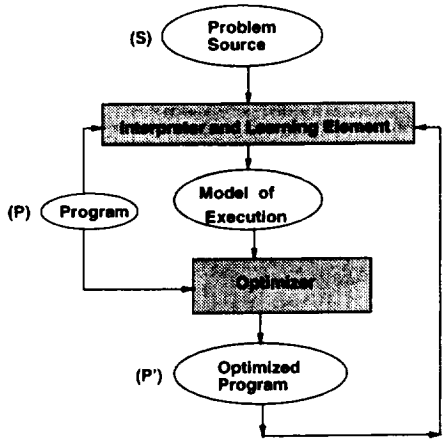

Figure 11: Dynamic Optimization Method

likely to improve performance. Applying these transformations to the program P results in a revised program P' equivalent to the original but with lower expected cost.

But we don't stop there: we repeat this learn/optimize process starting with P', deriving yet another optimized program P'', and so forth. To see why we need to iterate the improvement process, imagine that we subject the program in Figure 10(b) to the learn cycle and thereupon decide to reorder the clauses C_1' and C_2. As a result of this change the expected costs and probabilities for C_3 and C_4 will also change; hence any decision about the optimal order of C_3 and C_4 should be deferred until C_1' and C_2 have been reordered and the statistics revised.

In general, the cycle of learning and optimizing is repeated, with transformations occurring at successively deeper levels of the program, until no further optimizations can be found or some resource bound is exhausted. Unrolling *per se* has no effect on program performance but enables us to effect optimizations at specific points in the computation. Looking once again at Figure 10(b), we note that unrolling the predicate `p` into `p` and `pcopy` allows us to apply different transformations to the first call to `p` (clauses $C'1$ and C_2) and to the subsequent calls (clauses C_3 and C_4). As the program is progressively unrolled, the total number of clauses increases, but the physical size of the program is a negligible part of the run time.

The optimization technique described here is specific to Prolog in the choice of program transformations, but the same learn/optimize method applies to a broad family of languages—specifically, nondeterministic typed-term languages (Laird and Gamble, 1990), including lambda-calculus and combinator based languages. Imperative languages like C probably will benefit little from dynamic optimization, since the nondeterministic element (search) is not represented explicitly in the language.

As expected, the experimental results depended on both the program and the distribution of problems. On the one hand a program for parsing a context-free language ran more

Program	Average Improvement (%)	
	CPU Time	Unifications
CF Parser	41.1	34.5
List Membership	18.5	17.2
Logic Circuit Layout	4.8	9.5
Graph 3-Coloring	0.20	−1.40

Figure 12: Sample Dynamic program optimization results. Cost of computation was measured both by the count of unifications and by the total CPU time.

than 40% faster as a result of dynamic optimization; this was mainly the result of unfolding recursive productions that occurred with certainty or near certainty during the parsing. On the other hand a graph-coloring program implementing a brute-force backtracking search algorithm was not expected to benefit much from optimization—and, indeed, no improvement was obtained. Significantly, however, *little or no performance degradation was observed either.* Typical were speedups in the 10% to 20% range—entirely satisfactory in a production application. See Figure 12 for sample results.

In general, the TDAG-based method enjoys a number of advantages over other approaches:

- the ability to optimize general programs. Speedup methods based on explanation-based generalization do poorly on recursive programs.
- the ability to combine different program transformations in the same optimizer. Explanation-based learning methods are based only upon unfolding.
- absence of "generalization-to-N" anomalies and similar effects that result when program changes are based on a statistically insufficient sample of problem instances. TDAG collects statistics first and then changes the program only if the statistics predict a high likelihood of improvement.
- a robustness demonstrated by the fact that repeated runs of the optimizer on the same program have resulted in essentially the same optimized version.
- the fact that the order of the examples has little influence on the final optimized program. Partial evaluation and explanation based methods usually produce highly variable results depending on the order in which the examples are presented.

On the other hand, this approach will not improve the time complexity of the program by more than a constant factor. Nor can we offer any formal proof that this method will find an "optimal" or nearly optimal variant of the original program.

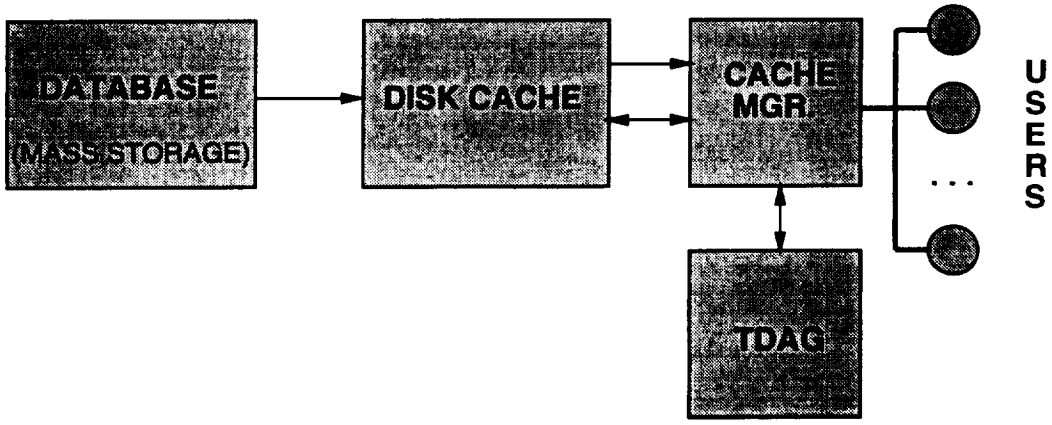

Figure 13: Predictive caching using TDAG

Predictive Caching in Mass Storage Systems. Mass storage systems (MSSs) holding terabytes worth of data are now commercially available, and larger ones are under development. In read-only form they serve as repositories for huge encyclopedias of data, but before the data can be processed, the relevant segments need to be copied into high-speed cache memory—an operation that can take several minutes. Since the cache can hold only a comparatively small number of segments, a cache manager (Figure 13) decides which segments to retain and which to discard when a user requests a segment not on the disk. The following model is based on specifications for a digital-tape MSS currently under commercial development.

Requests from users are directed to a central cache manager. The cache manager's objective is to minimize the total expected cost to its users. A user's cost is measured by the time spent waiting for data to be input and may be weighted by a priority factor reflecting the urgency of the user's task. Specifically, in our model the cost of failing to have a segment S resident in cache at demand time is the product of the time $T(S)$ to fetch S and the priority $W(U)$ assigned to the user U making the request. (For a tape medium the expected time to fetch S depends very little on the state of the hardware at the time of the request.) If $\Pr_U(S)$ denotes the likelihood that S will be the next segment requested by user U, the expected cost $C(S)$ incurred by not having S in cache at the time the next request arrives can be estimated by

$$C(S) = \sum_U \Pr_U(S) T(S) W(U). \tag{1}$$

Note that we are ignoring the effect of different arrival rates of requests from the users.

Each user is assumed to generate requests according to some stochastic process, independent of the requests of other users. Our model allows users to sign on and off, and each user session is assumed to be independent of previous activity by the same user. If the cache manager can predict which segment each user will request next, it can avoid discarding a

segment before it is needed. Moreover it can read in a segment in anticipation of its actual request if the prediction probability is sufficiently high. Given a prediction $Pr_U(S)$ for each user's next request, the cache manager can therefore optimize the cache by placing in its (unlocked) slots precisely the set of segments that minimizes $\sum_{S \notin \text{cache}} C(S)$.

Most memory management algorithms devised for virtual storage systems apply very primitive algorithms for predicting future user requests, typically by guessing that the segment least recently used (LRU) is the least likely to be used next. Anticipatory fetching has been suggested in the literature on virtual storage and tried in some studies, but has not found much use except for the heuristic that a request for a file block is likely to be followed by a request for the next consecutive block. For databases, mass storage and multiprocessor file systems, however, where the real-time constraints are less severe, more sophisticated prediction algorithms are feasible, and anticipatory fetching has improved system performance, e.g., (Kotz and Ellis, 1992; Palmer and Zdonik, 1991; Salem, 1991). In recent work, Vitter and Krishnan have analyzed prefetching algorithms and found conditions whereby a Ziv-Lempel compression algorithm, when used to predict the sequence of requests for a single user, is asymptotically optimal as a competitive online algorithm (Vitter and Krishnan, 1991). Experimental work applying Ziv-Lempel and other compression-related prediction techniques to database query processing is currently being conducted at several sites.

We performed some basic simulations to see how TDAG might perform relative to a simple LRU prediction algorithm in the cache manager of the MSS described above. Predictive caching was not part of the industrial design specification we obtained for the MSS; nevertheless we could not resist testing the effects of predictive caching using TDAG. Since these were simulations and not experiments using an actual MSS, times given in "seconds" represent arbitrary time units rather than actual machine times.

In our simulations, the cache manager builds a separate TDAG for each user so as to learn to predict that user's requests. For simplicity all users are assigned equal priorities. Each user is allocated one slot in the cache to hold the segment currently in use; this slot is locked so that this active segment will not be replaced. In addition, a pool of unlocked slots is available for storing segments likely to be used in the future. The cache manager tries to keep the cache in a state that minimizes the expected cost of the next request. When the current cache configuration differs from the predicted optimal configuration, the manager prefetches segments not in cache. Should a user demand a segment not present in cache, any outstanding prefetch is aborted, and the manager fetches the requested segment, placing it in the slot containing the segment with the lowest predicted cost. No direct cost is incurred for prefetching a segment that is not subsequently demanded, but the statistics below demonstrate the number of prefetches that proved useless.

The specifics of one experiment are as follows. Our simulated cache manager was given a pool of 12 slots. Competing for thoses slots were a variable number of active users making requests from a set of thirty segments. The users shared the available segments equally. We set up three scenarios: ten users each using 3 segments; six users using five segments; and two users using 15 segments. Each segment was assigned an arbitrary fixed transfer time between 5 and 20 seconds, representing the time required to move that segment from

mass storage to the cache. After demanding a segment, users spent roughly an order of magnitude longer "processing" the segment than was required to fetch the segment; thus the cache manager had sufficient time to manipulate contents of the free slots. Each user was modeled by a randomly generated stochastic finite automaton with between 9 and 30 active states and an out-degree of at most 3.

The tests whose data are given below consisted of two equal-length runs of tens of thousands of individual segment requests generated by the same SDFA user models. The purpose of such unrealistically long runs was to ensure the statistical significance of the results. In the first run the cache was managed with only an LRU strategy (TDAG prediction was not used). During this run a TDAG observed the requests and built a TDAG model of each user. The second run was then started in which the trained TDAGs were allowed to make predictions while the cache manager computed the cache configuration having the lowest expected waiting time, issuing prefetch requests as needed. The resulting statistics estimate the asymptotic difference between pure LRU caching and predictive caching based on trained TDAGs, free of the non-stationary effects of incremental learning. Figure ?? summarizes the test results. We repeated the experiments several times and always obtained similar results.

We also ran other tests in which the TDAGs learned and predicted simultaneously. We found that TDAG acquires very accurate predictions of the users' request patterns in roughly several hundred training instances. These experiments and others using a variety of user models and parameters are reported in (Laird and Saul, 1992).

The most straightforward measure of performance is the number of segment faults the users encountered during a run. The numbers in parentheses indicate the *miss ratio*: the total number of faults divided by total number of requests. We also include the percentage decrease in the number of faults as a result of using TDAG prefetching.

The values given for the mean waiting time are in (simulated) seconds per user request. The percent reduction in waiting time is especially instructive, since this is the actual quantity that our cache manager was trying to minimize, using equation (1).

Note that with 10 users there are only 2 free cache slots to manipulate, making this scenario the most difficult in which to achieve good performance. Obviously, both strategies benefit from having more free slots, and this is apparent across our three cases.

Also shown in the table is the number of prefetches issued during the second run. The sum of prefetches and faults divided by the total requests is often called the *transfer ratio*, that is, the number of physical reads issued per user request. In the absence of prefetching the transfer ratio equals the miss ratio; with prefetching, it is expected to be higher than the miss ratio, but our results show a large rise when there are but two users. The main reason is that our model charged no cost for issuing a prefetch—an assumption that is not unreasonable for dedicated servers with interruptible DMA hardware. Thus, in these cases, when many of the cache slots were unlocked and available, our cache manager had free reign to swap in new contents as it saw fit, and a large number of these prefetches proved unnecessary. Clearly, therefore, a system in which every read incurs positive cost might need to modify this policy. With TDAG this is easy: increase the lower bound on the prediction probability for any prefetched segment.

	Case 1	Case 2	Case 3
Number of Users	10	6	2
Segments per User	3	5	15
Total Requests per Run	30000	30000	30000
Mean Waiting Time (LRU)	15.21	10.29	5.63
Mean Waiting Time (TDAG)	11.74	3.80	0.44
Waiting Time Reduction	23%	63%	92%
Demand Faults (LRU)	29948 (.998)	23291 (.78)	12886 (.43)
Demand Faults (TDAG)	25189 (.84)	9945 (.33)	1344 (.05)
Prefetch Fault Reduction	16%	57%	90%
Prefetches Issued	7408	18923	71714
Prefetch Transfer Ratio	1.09	.96	2.44

Figure 14: Experimental results with multiple users and 12-slot cache

TDAG can be used to improve caching in ways other than those we devised for our experiments. Whereas we used the TDAG to learn to predict the complete sequence of segment requests, others have studied prefetching by learning only the segment references that produce faults, i.e., a *fault* on segment x predicts a later *fault* on segment y (Lau, 1982; Martinez, 1982). The cost model we used to control the cache required learning the complete sequence, but a TDAG could as easily be asked to learn the sequence of faults. Another possibility is to use TDAG to predict *relative segment numbers*—e.g., to predict that the next segment number will be $\pm n$ greater than the current segment number, with a probability distribution over the values of $n \neq 0$. A special case of this that has been applied (Smith, 1978) is prefetching under the assumption of sequentiality, whereby a reference to segment n is a strong predictor of a subsequent reference to segment $n+1$. The recent work of Salem (Salem, 1991) extends this to more general predictabilities and is similar to using a TDAG with $\mathbf{H} = 1$.

Finally we note that TDAG may not be the prediction method of choice in all situations. For example, we were able to construct a sequence of segment requests favorable to the LRU strategy and on which the TDAG-based prefetcher performs quite badly. Evidently the designers of caching systems must decide empirically which models of request sequences are best under the given conditions, but the TDAG provides a simple model that performs well for many request processes.

Conclusions

Sequence prediction is a fundamental learning problem, with a number of variants. The TDAG algorithm for the Discrete Sequence Prediction problem is easy to implement and reason about, requires little knowledge about the input stream, has fast turnaround time,

uses little storage, is mathematically sound, and has worked well in three prototype applications.

TDAG is intended for online applications where the learning algorithm must keep pace with the input stream of observations. It is effective when the next symbol can be predicted based on the k preceding ones, where k may vary with the context.

For most applications, the TDAG procedure will not be called simply as a black-box subroutine; instead, the basic TDAG algorithm will be modified to include additional data at the nodes. This was the case for both the dynamic optimization and caching applications.

Besides exploring new applications, we anticipate that future research directions will go beyond the current rote learning of high-likelihood sequences by generalizing from strings to patterns. For example, the sequence "aac" may not occur very often, nor "abc" or "acc", but by generalizing the middle symbol into a character variable x, we can predict c reliably from the context "ax...". It is not difficult to modify the TDAG algorithm for this purpose, with the result that the TDAG becomes a directed acyclic graph (DAG) instead of a tree.

Handling these kinds of generalizations formally as acyclic regular expressions is a promising research direction.

Whereas the TDAG prediction algorithm is purely statistical, other methods of prediction are known for sequence extrapolation and time-series analysis. Effective combinations of these methods would be valuable extensions to a growing machine-learning toolbox.

Acknowledgments

Much of this work was done during the first author's stay at the Machine Inference Section of the Electrotechnical Laboratory in Tsukuba, Japan. Thanks to the members of the laboratory, especially to Dr. Taisuke Sato. Thanks also to Wray Buntine, Peter Cheeseman, Oren Etzioni, Peter Friedland, Yaron Gold, Paul Howard, Wayne Iba, Smadar Kedar, P. Krishnan, Steve Minton, Andy Philips, Armand Prieditis, Jeff Vitter, and Monte Zweben for suggestions. Peter Norvig generously supplied us with his elegant Prolog for use in the dynamic optimization research. Three anonymous reviewers also offered their useful insights.

This work was supported in part by the National Science Foundation (INT-9008726)

References

(Abe and Warmuth, 1990) N. Abe and M. Warmuth. On the computational complexity of approximating distributions by probabilistic automata. In *Proc. 3rd Workshop on Computational Learning Theory*, 1990.

(Bell *et al.*, 1990) T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, N.J., 1990.

(Blumer, 1990) A. Blumer. Application of DAWGs to data compression. In A. Capocelli,

(Dietterich and Michalski, 1986) T. Dietterich and R. Michalski. Learning to predict sequences. In R. S. Michalski et al., editor, *Machine Learning: An AI Approach, Vol. II.* Morgan Kaufmann, 1986.

(Gooley and Wah, 1989) M. Gooley and B. Wah. Efficient reordering of Prolog programs. *IEEE Trans. on Knowledge and Data Engineering*, 1:470–482, 1989.

(Gratch and DeJong, 1992) J. Gratch and G. DeJong. An analysis of learning to plan as a search problem. In *Proceedings of the Ninth International Machine Learning Conference*

Morgan Kaufmann, 1992.

(Greiner and Orponen, 1991) R. Greiner and P. Orponen. Probably approximately optimal derivation strategies. In *Proceedings 2nd International Conference, Knowledge Representation and Reasoning*, pages 277–288, 1991.

(Kotz and Ellis, 1992) D. Kotz and C. S. Ellis. Practical prefetching techniques for multiprocessor file systems. *Distributed and Parallel Databases*, 1:33–51, 1992.

(Laird and Gamble, 1990) P. Laird and E. Gamble. Extending EBG to term-rewriting systems. In *Proceedings AAAI-90.* American Association for Artificial Intelligence, 1990.

(Laird and Saul, 1992) P. Laird and R. Saul. Predictive caching using the TDAG algorithm. Technical Report FIA-92-30, NASA Ames Research Center, AI Research Branch, 1992.

(Laird, 1988) P. Laird. Efficient unsupervised learning. In D. Haussler and L. Pitt, editors, *Proceedings, 1st Comput. Learning Theory Workshop.* Morgan Kaufmann, 1988.

(Laird, 1992a) P. Laird. Discrete sequence prediction and its applications. In *Proc., 9th National Conference on Artificial Intelligence.* AAAI, 1992.

(Laird, 1992b) P. Laird. Dynamic optimization. In *Proc., 9th International Machine Learning Conference.* Morgan Kaufmann, 1992.

(Lau, 1982) E. J. Lau. Improving page prefetching with prior knowledge. *Performance Evaluation*, 2(3):195–206, 1982.

(Lelewer and Hirschberg, 1987) D. Lelewer and D. S. Hirschberg. Data compression. *ACM*

(Lindsay *et al.*, 1980) R. Lindsay, B. Buchanan, and *et al. DENDRAL.* McGraw-Hill, New York, 1980.

(Martinez, 1982) M. Martinez. Program behavior prediction and prepaging. *Acta Informatica*, 17:101–120, 1982.

(Norvig, 1991) P. Norvig. *Paradigms of A.I. Programming: Case Studies in Common LISP.* Morgan Kaufmann, 1991.

(Palmer and Zdonik, 1991) M. Palmer and S. B. Zdonik. Fido: a cache that learns to fetch. In *Proceedings of 17th International Conference on Very Large Data Bases*, 1991.

(Prieditis and Mostow, 1987) A. Prieditis and J. Mostow. Prolearn: Towards a Prolog interpreter that learns. In *Proceedings of AAAI-87.* Morgan Kauffman, 1987.

(Salem, 1991) Kenneth Salem. Adaptive prefetching for disk buffers. Technical Report Tr-91-46, University of Maryland and CESDIS, Goddard Space Flight Center, 1991.

(Sejnowski and Rosenberg, 1987) T. Sejnowski and C. Rosenberg. Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145-168, 1987.

(Smith, 1978) A. J. Smith. Sequentiality and prefetching in database systems. *Transactions on Database Systems*, 3(3):223–247, 1978.

(Subramanian and Feldman, 1990) D. Subramanian and R. Feldman. The utility of EBL in recursive domains. In *Proceedings, AAAI-90*, pages 942–949. American Association for Artificial Intelligence, 1990.

(Vitter and Krishnan, 1991) J. Vitter and P. Krishnan. Optimal prefetching via data compression. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, 1991.

(Williams, 1988) R. Williams. Dynamic history predictive compression. *Information Systems*, 13(1):129–140, 1988.